Parallel Implementation of a Semidefinite Programming Solver based on CSDP in a distributed memory cluster

Ivanov, I.D.; de Klerk, E.

Publication date:
2007

Document Version
Publisher's PDF, also known as Version of record

CentER

Discussion Paper

No. 2007–20

PARALLEL IMPLEMENTATION OF A SEMIDEFINITE PROGRAMMING SOLVER BASED ON CSDP ON A DISTRIBUTED MEMORY CLUSTER

By I.D. Ivanov, E. de Klerk

March 2007

ISSN 0924-7815

Tilburg University

Parallel implementation of a semidefinite programming solver based on CSDP on a distributed memory cluster

I.D. Ivanov* E. de Klerk†

March 1, 2007

Abstract

In this paper we present the algorithmic framework and practical aspects of implementing a parallel version of a primal-dual semidefinite programming solver on a distributed memory computer cluster. Our implementation is based on the CSDP solver and uses a message passing interface (MPI), and the ScaLAPACK library. A new feature is implemented to deal with problems that have rank-one constraint matrices. We show that significant improvement is obtained for a test set of problems with rank one constraint matrices. Moreover, we show that very good parallel efficiency is obtained for large-scale problems where the number of linear equality constraints is very large compared to the block sizes of the positive semidefinite matrix variables.

Keywords: Semidefinite programming, interior point methods, parallel computing, distributed memory cluster

AMS classification: 90C22, 90C51

JEL code: C60

1 Introduction

Semidefinite programming (SDP) has been a very popular area in mathematical programming since the early 1990's. Applications include LMI's in control theory, the Lovász ϑ-function in combinatorial optimization, robust optimization, approximation of maximum cuts in graphs, satisfiability problems, polynomial optimization by approximation, electronic structure computations in quantum chemistry, and many more (see *e.g.* [14, 6]).

There are several interior-point SDP solvers available such as SeDuMi [23] , CSDP [3], SDPT3 [25], SDPA [26], and DSDP5 [2]. Unfortunately, it is well-known that sparsity in SDP

*Faculty of Electrical Engineering, Mathematic and Computer Sciences, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands. I.D.Ivanov@tudelft.nl

†Department of Econometrics and OR, Tilburg University, Tilburg, the Netherlands. E.deKlerk@uvt.nl

data cannot be exploited as efficiently as in the linear programming (LP) case, and the sizes of problems that can be solved in practice are modest when compared to LP.

As a result, parallel versions of several SDP software packages have been developed like PDSDP [1], SDPARA [27] and a parallel version of CSDP [5]. The first two are designed for PC clusters using MPI[1] and ScaLAPACK[2] and the last one is designed for a shared memory computer architecture [5]. PDSDP is a parallel version of the DSDP5 solver developed by Benson, Ye, and Zhang [2] and it uses a dual scaling algorithm. SDPARA is a parallel version of SDPA for a computer cluster and employs a primal-dual interior-point method using the H..K..M. search direction, as does CSDP.

In this paper we present a open source version of CSDP for a Beowulf[3] (distributed) computer cluster. It makes use of MPI, and of the ScaLAPACK library for parallel algebra computations. The new software is meant to complement the parallel version of CSDP [5] that is already available for shared memory machines. A beta version of our software is freely available at

`www.st.ewi.tudelft.nl/~ivanov/csdp5.0gplR1.tar.gz`

Our implementation also has a new built-in capability to deal efficiently with rank-one constraint matrices. As a result, significant speedup is achieved when computing the Schur complement matrix for such problems. To the best of our knowledge our software is the first parallel primal-dual interior point solver to exploit rank-one structure. The only option available so far is to use SDPT3 (no parallel implementation available (yet)) or the dual scaling algorithm implemented in PDSDP.

Outline

This paper is organized as follows: in the next section we motivate our choice of computer architecture for our implementation. In Section 3 we discuss the algorithmic framework behind the solver. In particular, we give details about how a rank-one structure of data matrices is exploited. Section 4 describes details about the parallel part of the solver. Numerical test results for our code on benchmark problems are presented in Section 5. In Section 6 we make an efficiency comparison between our implementation and the solvers SDPARA and PDSDP that use the same computer architecture.

2 Choice of computer architecture

There are two main classifications of multiprocessor 'supercomputers' today with respect to the programming model, namely shared memory and distributed memory architectures. The two architectures are suited to solving different kinds of problems.

Shared memory machines are best suited to so-called 'fine-grained' parallel computing, where all of the pieces of the problem are dependent on the results of the other processes. Distributed memory machines on the other hand are best suited to 'coarse-grained' problems, where each node can compute its piece of the problem with less frequent communication.

[1] `http://www-unix.mcs.anl.gov/mpi/`
[2] `http://www.netlib.org/scalapack/`
[3] `http://www.beowulf.org/`

Another issue with distributed memory clusters is message passing. Since each node can only access its own memory space, there has to be a way for nodes to communicate with each other. Beowulf clusters use MPI to define how nodes communicate. An issue with MPI, however, is that there are two copies of data: one is on the node, and the other has been sent to a central server. The cluster must ensure that the data that each node is using is the latest.

Partitioning problems to solve them is the main difficulty with the Beowulf cluster. To run efficiently, problems have to be partitioned so that the pieces will run efficiently in the RAM, disk, networking, and other resources on each node. If nodes have a gigabyte of RAM but the problem's data set does not easily partition into pieces that run in a gigabyte, then the problem could run inefficiently. This issue with the dynamic load balancing is not a problem for shared memory computers.

The attraction to use Beowulf clusters lies in the low cost of both hardware and software and the control that builders/users have over their system. These clusters are cheaper, often by orders of magnitude, than single-node supercomputers. Advances in networking technologies and software in recent years have helped to level the field between massively parallel clusters and purpose-built supercomputers.
Ultimately the the choice between 'shared or distributed' depends on the problem one is trying to solve. In our case the parts of the SDP algorithm suitable for parallel computation are computing the Schur complement matrix and its Cholesky factorization.

First, composing the Schur complement matrix allows each node independently from the others to compute its piece of data and no communication is needed until the pieces of the matrix are assembled. This allows us to regard this computation as a course-grained process. Secondly, the ScaLAPACK routines employed for Cholesky factorization are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy.

3 Algorithmic Framework

We recall in this section the predictor-corrector primal-dual interior point algorithm implemented in the original code of CSDP [3]. Consider the semidefinite programming (SDP) problem formulation in primal form

$$\begin{aligned} (\mathrm{P}) \qquad \max_{X} \quad & \mathbf{Tr}(CX) \\ \text{s.t.} \quad & A(X) = b, \\ & X \succeq 0 \end{aligned} \tag{3.1}$$

where $X \succeq 0$ (or $X \in \mathcal{S}_n^+$) means that X is a symmetric positive semidefinite $n \times n$ matrix. We assume a representation of the linear operator A as

$$A(X) = \begin{bmatrix} \mathbf{Tr}(A_1 X) \\ \mathbf{Tr}(A_2 X) \\ \vdots \\ \mathbf{Tr}(A_m X) \end{bmatrix}, \tag{3.2}$$

where the matrices $A_i \in \mathcal{S}_n$ (*i.e.* are symmetric $n \times n$) for $i = 1, \cdots, m$, as is the matrix C. Thus n denotes the size of the matrix variables and m the number of equality constraints. The dual of the SDP problem (P) is given by

$$\begin{aligned} \text{(D)} \qquad & \min_{y,Z} \quad b^T y \\ \text{s.t.} \qquad & A^*(y) - Z = C, \\ & Z \succeq 0 \end{aligned}$$

where

$$A^*(y) = \sum_i^m y_i A_i,$$

is the adjoint of A with respect to the usual trace inner product. A primal X and dual (y, Z) are interior feasible solutions of (P) and (D), respectively, if they satisfy the constraints of (P) and (D) as well as $X \succ 0$ and $Z \succ 0$. (We use the notation $X \succ 0$ for $X \in \mathcal{S}_n$ to be positive definite.) The idea behind primal-dual IPM's is to 'follow' the central path

$$\left\{(X(\mu), y(\mu), Z(\mu)) \in \mathcal{S}_n^+ \times \mathbb{R}^m \times \mathcal{S}_n^+ : \mu > 0\right\}$$

where each $(X(\mu), y(\mu), Z(\mu))$ is a solution of the system of equations

$$\begin{aligned} b - A(X) &= 0, \\ Z + C - A^*(y) &= 0, \\ ZX - \mu I &= 0, \\ Z, X &\succ 0, \end{aligned} \tag{3.3}$$

where I denotes the identity matrix of size $n \times n$, and $\mu > 0$ is a given value called the *barrier parameter*.

It is well-known that for each $\mu > 0$ (3.3) has an unique solution $(X(\mu), y(\mu), Z(\mu))$ assuming that there exists a interior feasible solution (X, y, Z) of the SDP and constraint matrices $A_i \in \mathcal{S}_n$ for $i = 1, \cdots, m$ are linearly independent. Moreover, in the limit as μ goes to 0, $(X(\mu), y(\mu), Z(\mu))$ converges to an optimal solution of the SDP.

The algorithm used in CSDP is an *infeasible-start method* and it is designed to work with starting point $X \succ 0, Z \succ 0$ that is not necessarily feasible. An alternative approach used in some SDP solvers as SeDuMi [23] is to use a *self-dual embedding* [8] technique to obtain a feasible starting point on the central path. In our implementation we use the default CSDP starting point (a similar approach is used in [25]):

$$\begin{aligned} X &= \alpha I, \\ y &= 0, \\ Z &= \beta I \end{aligned} \tag{3.4}$$

where

$$\alpha = n \max_k \left(\frac{1 + \mid b_k \mid}{1 + \|A_k\|_F} \right) \text{ and } \beta = \frac{1}{\sqrt{n}} \left(1 + \max(\max_k(\|A_k\|_F, \|C\|_F)) \right).$$

It is easy to see that this initial point may not satisfy $A(X) = b$ or $A^*(y) - Z = C$, so we define

$$\begin{aligned} R_p &= b - A(X), \\ R_d &= Z + C - A^*(y). \end{aligned}$$

The algorithm implemented in CSDP uses a predictor-corrector strategy. The predictor step is computed from:

$$\begin{aligned} -A(\Delta\widehat{X}) &= -R_p, \\ \Delta\widehat{Z} + C - A^*(\Delta\widehat{y}) &= -R_d, \\ Z\Delta\widehat{X} - \Delta\widehat{Z}X &= -ZX. \end{aligned} \tag{3.5}$$

Using the same approach as in [14], we can reduce the system of equations (3.5) to

$$A(Z^{-1}A^*(\Delta\widehat{y})X) = -b + A(Z^{-1}R_dX), \tag{3.6}$$

$$\Delta\widehat{Z} = A^*(\Delta\widehat{y}) - R_d, \tag{3.7}$$

$$\Delta\widehat{X} = -X + Z^{-1}R_dX - Z^{-1}A^*(\Delta\widehat{y})X. \tag{3.8}$$

If we introduce the notation

$$\begin{aligned} M &:= [A(Z^{-1}A_1X), A(Z^{-1}A_2X), \ldots, A(Z^{-1}A_mX)], \\ v &:= -b + A(Z^{-1}R_dX), \end{aligned} \tag{3.9}$$

then we can write (3.6) in matrix form as follows:

$$M\Delta\widehat{y} = v. \tag{3.10}$$

As Helmberg, Rendl, Vanderbei and Wolkowicz have shown, the Schur complement matrix M is symmetric and positive definite [14]. Thus we can compute the Cholesky factorization of M to solve the system of equations (3.10). By back substitution in (3.7) and (3.8) we can subsequently compute $\Delta\widehat{Z}$ and $\Delta\widehat{X}$ respectively. Note that in this case $\Delta\widehat{X}$ is not necessarily symmetric. In order to keep $X + \Delta\widehat{X}$ symmetric, we replace $\Delta\widehat{X}$ by $\frac{\Delta\widehat{X}+\Delta\widehat{X}^T}{2}$.

For the corrector step we solve the linear system

$$\begin{aligned} -A(\Delta\overline{X}) &= 0, \\ \Delta\overline{Z} + C - A^*(\Delta\overline{y}) &= 0, \\ Z\Delta\overline{X} + \Delta\overline{Z}X &= \mu I - \Delta\widehat{Z}\Delta\widehat{X}, \end{aligned} \tag{3.11}$$

where $\mu = \mathbf{Tr}\frac{XZ}{2n}$.

These equations have the same form as (3.5) and are solved similarly as before to obtain $(\Delta\overline{X}, \Delta\overline{y}, \Delta\overline{Z})$. Next we add the predictor and corrector step to compute the search directions:

$$\begin{aligned} \triangle X &= \Delta\widehat{X} + \Delta\overline{X}, \\ \triangle y &= \Delta\widehat{y} + \Delta\overline{y}, \\ \triangle Z &= \Delta\widehat{Z} + \Delta\overline{Z}. \end{aligned} \tag{3.12}$$

Next, we find the maximum step lengths α_P and α_D such that the update $(X + \alpha_P \triangle X, y + \alpha_D \triangle y, Z + \alpha_D \triangle Z)$ results in a feasible primal–dual point.

In practice, the Schur complement matrix M may become numerically singular even though X and Z are numerically nonsingular. In this case, CSDP returns to the previous solution, and executes a centering step with $\mu = \mathbf{Tr}\frac{XZ}{n}$.

The default stopping criteria are the following

$$\begin{aligned} \frac{\mid \mathbf{Tr}(CX) - b^T y \mid}{1 + |b^T y|} &< 10^{-7}, \\ \frac{\|A(X) - b\|}{1 + \|b\|} &< 10^{-7}, \\ \frac{\|A^*(y) - Z - C\|_F}{1 + \|C\|_F} &< 10^{-7}, \\ Z, X &\succeq 0. \end{aligned} \tag{3.13}$$

The solution of the linear system (3.10) involves the construction and the Cholesky factorization of the Schur complement matrix $M \succ 0$ that has size m by m. For dense X, Z and A_i $i = 1, \ldots, m$, the worst case complexity in computing M is $\mathcal{O}(mn^3 + m^2n^2)$ [19]. In practice the constraint matrices are very sparse and we exploit sparsity in the construction of the Schur complement matrix in the same way as in [10]. For sparse $A_i's$ with $\mathcal{O}(1)$ entries, the matrix $Z^{-1}A_iX$ can be computed in $\mathcal{O}(n^2)$ operations $(i = 1, \ldots, m)$. Additionally $\mathcal{O}(m^2)$ time is required for $A(\cdot)$ operations. Finally, the Schur complement matrix M is typically fully dense and its Cholesky factorization requires $\mathcal{O}(m^3)$ operations.

There are results on exploiting aggregate sparsity patterns of data matrices via matrix completion [11]. Results from the practical implementation of this approach (see [20]) show that it is efficient only on a very sparse SDP's where the aggregate sparsity pattern of the constraint matrices induces a sparse chordal graph. If this is not the case, a general SDP solver will have better performance. Therefore, we didn't consider exploiting this structure in our implementation since our aim was not a problem specific solver.

The overall computational complexity of presented primal-dual IPM is dominated by different operations depending on the particular structure of the SDP problem. For problems with $m \gg n$ more CPU time is spent in computation of the $m \times m$ Schur complement matrix M, and computation of the solution $\Delta\widehat{y}$ of (3.10). When the constraint matrices are very sparse, factoring the $m \times m$ matrix M becomes the dominant operation. From now on we will refer to the Choleski factorization procedure as *Cholesky*. In case of $m \ll n$ and a a small number of big diagonal blocks then matrix operations on positive semidefinite X and Z can be the dominant ones. In general the primal variable X is fully dense even if all the constraint matrices A_i $(i = 1, ..., m)$

are sparse. On the other hand, the dual matrix variable Z computed by

$$Z = \sum_{i=1}^{m} y_i A_i - C,$$

inherits the aggregate sparsity of the constraint matrices A_i $(i = 1, ..., m)$ and C. Thus primal-dual IPM's are at a disadvantage when compared to dual interior-point methods which generate iterates only in the dual space. Despite its lower computational cost per iteration, the dual IPM's do not posses super-linear convergence rates and typically attain less accuracy in practice [27]. Later on in this section, we propose a modification in storing rank-one $A_i's$ and computing M when using a primal-dual IPM.

When we have dense constraints, construction of M, called *Elements* from now on, can be the most time consuming computation in each iteration when $m >> n$. To reduce the computational time of our code takes advantage of the of the approach by Fujisawa, Kojima, and Nakata [10]. As test results with SDPLIB suggest [5],[26] the most computational time in general large scale problems using primal-dual IPM algorithms is occupied by constructing *Elements* and solving the linear system which involves *Cholesky*. This motivated our work toward employing distributed parallel computation for these computations.

Exploiting rank-1 structure of constraint matrices

In this subsection one additional assumption will be made: the constraint matrices have the form $A_i = a_i {a_i}^T$, $a_i \in \mathbb{R}^n$ and $i = 1, ..., m$. We will refer to this type of structure as rank-one. It appears in many large scale problems coming from combinatorial optimization problems and optimization of univariate functions by interpolation [7]. We assume that all of the constraint matrices are rank-one. Our aim next is to use this special structure of $A_i's$ to speed up computation of *Elements* when using a primal-dual IPM. The approach we use is basically one introduced by Helmberg and Rendl [13]. Recall from (3.9) that

$$M_{ij} = \mathbf{Tr}(A_i Z^{-1} A_j X) \qquad (i, j = 1, \ldots, m).$$

Since $A_i = a_i {a_i}^T$, this reduces to

$$M_{ij} = ({a_i}^T Z^{-1} a_j)({a_i}^T X a_j) \qquad (i, j = 1, \ldots, m). \tag{3.14}$$

Precomputing ${a_i}^T Z^{-1}$ and ${a_i}^T X$ for each row leads to an $\mathcal{O}(mn^2 + m^2 n)$ arithmetic operations procedure for building M [13]. In practice this can be improved significantly for sparse a_i's.

We would like to store only the vectors a_i as opposed to the matrices $A_i = a_i a_i^T$ since this obviously reduces the storage requirements by a factor n.

On the other hand, we still want to use the standard SDPA sparse input data format, that does not have an option to store rank-one matrices efficiently. Since CSDP originally works with a block diagonal matrix structure, we save each vector $a_i, i = 1, ..., m$ as a diagonal matrix, and introduce an additional parameter that we call *rank1* into the file of CSDP input parameters *param.csdp*, to ensure that the constraint matrices are interpreted in the right way by our modified software (i.e. not as diagonal matrices, but as a rank-one matrices).

4 Parallel implementation of CSDP

In this section we describe our parallel version of CSDP on a Beowulf cluster of PC's. The code is based on CSDP 5.0 and it enjoys the same 64-bit capability as the shared memory version [5]. The code is written in ANSI C with additional MPI directives and use of ScaLAPACK library for parallel algebra computations. We also assume that optimized BLAS and LAPACK libraries are available on all nodes. The latter are used for implementation of matrix multiplication, Cholesky factorization and other linear algebra operations. As we already mentioned in the previous section, the most time-consuming operations are the computation of the elements of Schur complement matrix M and its Cholesky factorization. Therefore, in our development we used parallelization to accelerate these two bottlenecks in the sequential algorithm. Next we describe how our software actually works.

We denote by N the number of processors available, and we attach a corresponding process $P_j, j = 0, .., N-1$ to each one of them. We call process P_0 a *root*. When defined in this way, the processes form a one-dimensional array. To be able to use ScaLAPACK efficiently, it is useful to map this one-dimensional array of processes into a two-dimensional rectangular array, called often a process grid. Let this process grid have N_r rows and N_c columns, where $N_r * N_c = N$. Each process is thus indexed by its row and column coordinates as $P_{row,col}$, with $0 \leq row \leq N_r - 1$ and $0 \leq col \leq N_c - 1$.

The software starts execution by sending a copy of the execution and data files to all N nodes. Each node independently proceeds to allocate space for the variables X, y, Z locally, and starts the execution of its copy of the code, until computation of the Schur complement matrix M is reached.

We use a distributed storage for M during the complete solution cycle of the algorithm. As a result the software does not require that any of the nodes should be able to accommodate the whole matrix in its memory space. It has to be mentioned that M is stored on distributed memory only in one storage format, and not in two different formats as in SDPARA. We only use a two-dimensional block cyclic data distribution required later on from ScaLAPACK, see Figure 1. In particular, the matrix M is subdivided into blocks of size $NB \times NB$ for a suitable value of NB and the assignment of block to processes are as shown in the figure. Since the matrix M is symmetric, we need only to compute the upper triangular part of it.

For this two-dimensional block cyclic layout it is difficult to achieve a good load balance. The quality of the load balance depends on the choices of NB, N_r and N_c. If we choose $N_c = 1$ and $N_r = N$ we will have a very good load balance, *i.e.* we will have a one-dimensional block row distribution. On the other hand, the distributed Cholesky factorization of M performed by ScaLAPACK reaches its best performance when the process grid is as square as possible ($N_r \approx N_c$). As we will show later, the square process grid is justified when we use Beowulf clusters that use only GBit ethernet interconnect, which has relatively high latency ($50 - 100\mu s$) compared with Myrinet or Infiniband interconnects ($6 - 9\mu s$).

5 Numerical results

In this section we present numerical results testing our rank-one feature on the sequential CSDP code. We also show results in the last subsection from tests of our parallel implementation

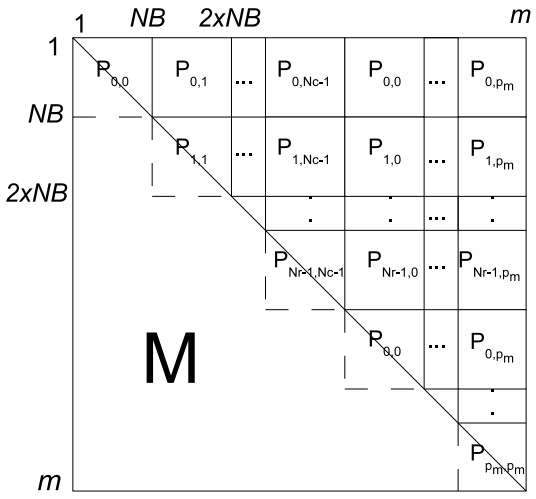

Figure 1: Two-dimensional block cyclic data distribution over two dimensional process grid.

on a number of selected SDP test problems taken from SDPLIB, DIMACS and other sets of benchmark problems.

5.1 Experimental results for the sequential code on rank-one problems

In this section we test the practical efficiency of our modified sequential CSDP code to deal with rank-one constraints. From now on we will recall this version as CSDP-R1. The numerical experiments were executed on PC with Intel x86 P4 (3.0GHz) CPU and 3GB of memory running Linux operation system. An optimized BLAS and LAPACK libraries were used. Both codes were compiled with gcc 3.4.5 the default values of all parameters from CSDP 5.0[4] were used. The input data format used is the standard SDPA sparse (dat-s) format for CSDP and our modified SDPA sparse format for CSDP-R1.

Optimization of univariate functions on bounded interval by interpolation

The performance of both CSDP and CSDP-R1 was tested on SDP instances from [7]. These SDP problems approximate the minima of a set of twenty univariate test functions from Hansen et al. [12] on an interval, by first approximating the functions using Lagrange interpolation, and subsequently minimizing the Lagrange polynomials using SDP. These SDP problems only involve rank-one data matrices by using a formulation due to Löfberg and Parrilo [16].

The data in Tables 1 and 2 describes the results from our experiments with respect to the number of (Chebyshev) interpolation points 20, 40, 60 80, 100, 150 and 200 for CSDP and CSDP-R1, respectively. The sizes of the SDP problems roughly depend on the number of interpolation points as follows: the number of linear equality constraints m equals the number of interpolation points, and there are two positive semidefinite blocks of size roughly $m/2$.

[4]http://infohost.nmt.edu/ borchers/csdp.html/

Problem	Objective	20	40	60	80	100	150	200
$test1$	$2.9763233e+04$	0.05	0.58	2.30	7.01	16.05	76.87	243.78
$test2$	$1.8995993e+00$	0.06	0.62	2.30	7.16	16.19	78.38	250.86
$test3$	$1.2031249e+01$	0.06	0.58	2.46	7.32	15.54	80.11	235.20
$test4$	$3.8504507e+00$	0.05	0.56	2.21	6.68	15.75	73.06	251.04
$test5$	$1.4890725e+00$	0.06	0.62	2.37	7.40	16.88	76.90	244.48
$test6$	$8.2423940e-01$	0.06	0.57	2.31	7.03	16.87	82.48	254.22
$test7$	$1.6013075e+00$	0.06	0.62	2.38	7.44	18.13	85.20	258.01
$test8$	$1.4508008e+01$	0.05	0.60	2.34	7.19	15.78	78.82	254.26
$test9$	$1.9059611e+00$	0.05	0.64	2.43	7.41	16.92	79.42	255.00
$test10$	$7.9167274e+00$	0.06	0.62	2.45	7.62	17.20	76.33	245.32
$test11$	$1.5000000e+00$	0.05	0.62	2.50	7.86	17.06	82.95	265.09
$test12$	$1.0000000e+00$	0.07	0.61	2.37	7.65	17.02	80.48	254.71
$test13$	$1.5874011e+00$	0.06	0.62	2.42	7.33	16.26	79.07	251.49
$test14$	$7.8868539e-01$	0.06	0.61	2.38	7.45	16.95	80.61	261.72
$test15$	$3.5533901e-02$	0.07	0.73	2.80	8.40	19.15	86.17	272.15
$test16$	$-1.110901e-02$	0.07	0.76	3.03	9.07	21.50	98.18	321.54
$test17$	$-7.0000000e+00$	0.06	0.76	2.88	8.84	20.71	101.68	300.42
$test18$	0	0.07	0.72	2.59	8.45	18.58	89.28	267.22
$test19$	$7.8156745e+00$	0.07	0.61	2.48	7.53	17.03	82.57	264.96
$test20$	$6.3490529e-02$	0.07	0.58	2.36	6.94	17.24	79.33	239.42

Table 1: Solution times in seconds for CSDP for twenty rank-one test problems from [7].

We also give the optimal objective value for each test function.

The solution times in Tables 1 and 2 show that when we have up to 40 interpolation points, the times are of the same order. This is to be expected due to the small size of the SDP problem. With increasing the number of interpolation points, the difference in solution times becomes apparent. When the order is 60, we notice roughly a 50% reduction in solution time when exploiting rank-one structure. This percentage increases with increase of the number of interpolation points. For order 100, CSDP-R1 obtains solution almost twice faster on average than the standard version of CSDP. The difference in solution time is almost 3 times in favor of CSDP-R1 for 200 interpolation points. These results clearly indicate that the simple construction (3.14) for exploiting rank-one structure when computing *Elements* in a primal-dual IPM algorithm results in a very significant speedup in practise.

Next, we included in our tests the fastest known approach to solve rank-one problems, namely using the DSDP solver. We performed tests with DSDP using its Matlab 6.5 interface and Linux operating system on the same PC as the test of CSDP and CSDP-R1. The test was only on the test functions test1 and test17 functions, as the computational times vary only slightly for the different instances. The results are shown in Table 3. All times are in seconds and we were only interested in the total running time. We see that for order 20 and 40 we have a running

Problem	Objective	20	40	60	80	100	150	200
*test*1	$2.9763233e+04$	0.08	0.40	1.36	3.44	7.59	31.58	92.90
*test*2	$1.8995993e+00$	0.06	0.42	1.38	3.46	7.51	31.96	98.28
*test*3	$1.2031249e+01$	0.07	0.39	1.37	3.57	7.14	30.47	98.08
*test*4	$3.8504507e+00$	0.06	0.39	1.28	3.36	7.25	31.22	88.95
*test*5	$1.4890725e+00$	0.07	0.43	1.36	3.59	7.60	31.18	94.78
*test*6	$8.2423940e-01$	0.07	0.40	1.38	3.47	7.63	33.74	93.25
*test*7	$1.6013075e+00$	0.07	0.42	1.37	3.64	7.63	34.62	96.39
*test*8	$1.4508008e+01$	0.06	0.39	1.34	3.66	7.14	30.48	92.77
*test*9	$1.9059611e+00$	0.07	0.42	1.35	3.55	7.50	32.36	93.95
*test*10	$7.9167274e+00$	0.07	0.40	1.33	3.47	7.71	32.38	92.15
*test*11	$1.5000000e+00$	0.07	0.40	1.34	3.65	7.49	34.43	97.44
*test*12	$1.0000000e+00$	0.06	0.39	1.25	3.64	7.15	31.96	99.44
*test*13	$1.5874011e+00$	0.06	0.38	1.26	3.51	7.35	30.93	92.60
*test*14	$7.8868539e-01$	0.06	0.39	1.31	3.52	7.33	33.03	95.05
*test*15	$3.5533901e-02$	0.07	0.45	1.51	4.08	8.54	35.10	102.84
*test*16	$-1.110901e-02$	0.08	0.54	1.69	4.64	9.65	41.63	112.57
*test*17	$-7.0000000e+00$	0.07	0.48	1.56	4.23	8.69	37.34	108.08
*test*18	0	0.08	0.49	1.65	4.04	8.72	35.01	98.89
*test*19	$7.8156745e+00$	0.06	0.39	1.35	3.61	7.53	32.02	93.05
*test*20	$6.3490529e-02$	0.06	0.38	1.28	3.53	7.66	30.88	94.87

Table 2: Solution times in seconds for CSDP-R1 for twenty rank-one test problems from [7].

times with difference within a 2-3 tenths of a second for DSDP, CSDP and CSDP-R1. When the order is 100 then DSDP is faster than CSDP-R1 by a factor of three. When the size of the SDP problem further increases, the gap between the two algorithms grows as one might expect. For 200 interpolation points, CSDP-R1 is slower than DSDP by a factor close to 10.

In summary, from all our numerical experiments so far exploiting rank-one structure in CSDP still does not compete well with the dual scaling algorithm implemented in DSDP, but it does make the gap smaller than before.

5.2 Experimental results on parallel code

In this section we present results on numerical tests of our parallel implementation of CSDP using MPI, which we will refer to as PCSDP. The software was developed and tested on the DAS3 Beowulf cluster at the Delft University of Technology. Each node of the cluster has two 64bit AMD Opteron 2.4 GHz CPU, running ClusterVisionOS Linux, and has 4 GB memory. Communication between the nodes relies on GBit ethernet network, which is used as an interconnect and network file transport through a 10GBit switch. All parameter values were set to the default values for CSDP, excluding the new *rank1* parameter.

Problem	Solver	time	20	40	60	80	100	150	200
test1	CSDP	Reading input	0.01	0.08	0.28	0.64	1.28	4.38	10.69
		Solver	0.05	0.58	2.30	7.01	16.05	76.87	243.78
		Total	0.06	0.66	2.58	7.65	17.33	81.45	254.47
	CSDP-R1	Reading input	< 0.01	0.01	0.02	0.02	0.04	0.11	0.22
		Solver	0.08	0.4	1.36	3.44	7.59	31.58	92.90
		Total	0.08	0.41	1.38	3.46	7.63	31.69	93.12
	DSDP	Total	0.15	0.34	0.76	1.27	2.48	6.75	14.80
test17	CSDP	Reading input	0.01	0.09	0.28	0.67	1.31	4.43	10.50
		Solver	0.06	0.76	2.88	8.84	20.71	101.68	300.42
		Total	0.07	0.85	3.16	9.51	22.02	106.11	310.92
	CSDP-R1	Reading input	< 0.01	0.01	0.02	0.02	0.05	0.11	0.20
		Solver	0.07	0.48	1.56	4.23	8.69	37.34	108.08
		Total	0.07	0.49	1.58	4.25	8.74	37.45	108.28
	DSDP	Total	0.08	0.20	0.44	0.88	1.49	4.78	11.19

Table 3: Running times in seconds for CSDP, CSDP-R1 and DSDP for one rank-one test problem from [7].

As is customary, we measured speedups S parallel efficiencies E, defined by

$$S = \frac{T_1}{T_N} \quad \text{and} \quad E = \frac{S}{N} = \frac{T_1}{(NT_N)}, \tag{5.15}$$

where T_1 and T_N are the times to run a code on one processor and N processors respectively. Note that $0 \leq E \leq 1$ and $E = 1$ corresponds to perfect speedup.

Results on SDP benchmark problems

We selected for our tests sixteen medium and large-scale SDP problems from five different applications. These applications are: control theory, the maximum cut problem, the *Lovász* ϑ-function, the min k-uncut problem, and calculating electronic structures in quantum chemistry. All of the test instances are from a standard benchmark suites. More details about them can be found in Table 9.

Problems control10 and control11 are from control theory and they are the largest of this type in SDPLIB test set [4]. The instance maxG51 is a medium size max-cut problem chosen from the same benchmark set. Theta5, theta6, thetaG51, theta42, theta62, theta8, and theta82 are *Lovász* ϑ problems. The first three are from SDPLIB, while the others were generated by Toh and Kojima [24]. The instances hamming_8_3_4, hamming_9_5_6, hamming_10_2 and hamming_11_2 compute the ϑ function of Hamming graphs. All four of them are from the DIMACS [17] library of mixed semidefinite-quadratic-linear programs as is the min k-uncut test problem fap09. The sixteenth instance is CH4.1A1.STO6G, that comes form calculating

electronic structures in quantum chemistry [21]. The initial point used during the numerical experiments is as described in (3.4) and the stopping criteria is (3.13).

Tables 4 and 5 give the time in seconds for computing Schur complement matrix M, Cholesky factorization of M and total solution time using one to sixty four processors.

In these tables m is the number of constraints as before, and n_{max} is the size of the largest block in the primal matrix variable X. Computing *Elements* (the Schur complement matrix) for control theory problems control10 and control11 is the most time consuming operation. For the rest of problems except maxG51, the time to compute the Cholesky factorization of M dominates. The parallel efficiencies were calculated using (5.15). This was not possible for all of the test problems, though: when the number of constraints exceed $20,000$, the memory required to accommodate the problem exceeded the available memory on an individual node. Therefore, Table 4 presents only the parallel efficiencies of the problem that can run on 1 node.

For problems control10 and control11, computing the Schur complement matrix is the dominant task and it scales relatively well with the number of the processors. However this is not the case with Cholesky factorization of M and the total running time, especially when the number of processors is 16, 32 and 64. The $m \times m$ Schur complement matrix is fully dense in general and its Cholesky factorization does not inherit much from the structure of the problem. Although the block structure and the sparsity of X, Z and A_i are effectively utilized in the matrix multiplications, they do not affect the scalability of distributed computing of *Cholesky*. Additionally, the high latency of the GBit network interconnect between the nodes increases the time each message is delivered. Therefore, when the problem has fewer than $10,000$ constraints, like control10 and control11, it is not very efficient to solve it by distributed memory cluster. The same issue is observed also in the middle-sized *Lovász* ϑ-type problems theta42, theta5, theta6 and theta8. The parallel efficiency in these cases drop under 0.5 when more than 16 processors are used. The story is similar for the problem thetaG51. This is to be expected due to the relatively large dimensions of the primal and dual matrix variables X and Z.

Another example that sufferers from low parallel-to-nonparallel ratio is the max-cut problem maxG51, where $n = m$. The constraints itself are very sparse (and rank-one). As a result computing the elements of M and its Cholesky factorization takes a very small part in the total solution time. The dominant operations in this case are factorization of the primal matrix variable. Therefore, the overall scalability is poor and solving such a problem in a distributed memory computers is very inefficient. Similar problem with the scalability of this problem was observed in [27].

Numerical results so far clearly indicate that problems with number of constraints under $10,000$ and small n or ones with $m = n$ are not solved very efficiently on more than 4 processors by our distributed memory cluster.

Much better results are obtained for the large scale instances as theta62, fap09 and hamming_8_3_4. Here the parallel efficiency is above 0.5 for almost the whole range of CPU's used between 2 and 64. Only for 32 it is slightly lower (0.47 due to the non squire grid geometry in this case i.e. $N_r = 8, N_c = 4$). Both the Cholesky factorization and construction of the Schur complement matrix scale well with the number of processors for all the three problems. When these two computations dominate in the overall running time one sees a good overall scalability of the problem.

As we already mentioned above, we were not able to compute the parallel efficiencies for the

Problem	m	n_{max}	*Phase*	1	2	4	8	16	32	64
control10	1326	100	Elements	280.35	144.66	77.7	43.98	31.72	21.17	12.32
			Cholesky	34.57	24.15	19.52	15.58	10.73	9.57	8.26
			Total	344.00	196.01	124.39	87.82	71.62	62.33	56.61
control11	1596	110	Elements	489.53	243.16	125.00	71.03	53.09	30.57	21.66
			Cholesky	74.61	40.17	30.98	23.48	15.09	14.5	9.69
			Total	605.29	310.55	191.96	129.84	106.94	83.34	75.30
theta5	3028	250	Elements	13.87	11.48	6.64	3.71	2.53	1.45	1.27
			Cholesky	192.41	99.58	57.03	39.68	23.02	19.73	11.44
			Total	229.96	129.12	79.77	57.28	38.8	33.9	25.67
theta6	4375	300	Elements	38.54	19.11	12.00	5.79	4.01	2.8	1.81
			Cholesky	590.16	310.97	160.9	102.41	56.68	40.59	25.25
			Total	696.77	367.71	202.21	122.93	83.68	75.75	49.94
theta8	7905	400	Elements	183.87	97.42	50.57	27.32	17.78	12.28	9.98
			Cholesky	3336.99	2375.22	1276.52	524.33	274.71	239.74	100.14
			Total	3673.97	2592.06	1426.64	635.77	372.40	327.14	184.52
theta42	5986	200	Elements	88.18	43.46	22.75	12.14	7.93	4.23	3.82
			Cholesky	1489.70	827.11	452.74	226.59	122.00	107.09	48.65
			Total	1647.20	915.63	509.96	256.04	153.66	132.71	73.08
theta62	13390	300	Elements	508.96	256.00	150.42	72.20	43.81	21.62	14.85
			Cholesky	15094.69	13065.39	6790.26	2746.30	1422.07	923.26	355.69
			Total	15927.84	13530.76	7100.53	2935.48	1571.48	1034.38	459.62
thetaG51	6910	1001	Elements	971.22	487.91	247.11	124.44	66.65	35.05	20.96
			Cholesky	5258.25	3341.09	1852.00	826.88	436.09	369.94	165.19
			Total	7721.57	5368.77	3633.84	2498.57	2041.32	1943.69	1715.75
maxG51	1000	1000	Elements	1.29	1.10	0.62	0.28	0.19	0.12	0.08
			Cholesky	7.95	6.28	4.90	4.54	3.78	3.85	3.74
			Total	641.45	640.46	639.83	635.04	632.19	632.04	632.92
hamming_8_3_4	16129	256	Elements	629.04	309.19	186.58	96.20	57.52	24.44	19.32
			Cholesky	22980.47	15960.96	10340.12	5151.16	2278.29	1298.03	511.16
			Total	29237.21	16526.65	10719.20	5391.22	2462.25	1433.83	685.14
fap09	15225	174	Elements	6656.76	3548.41	2476.86	1226.86	869.74	413.42	335.49
			Cholesky	113529.39	97491.86	46902.41	23145.18	11928.57	7140.35	2714.18
			Total	122334.83	102429.12	50434.07	25155.29	13490.96	8153.83	3621.23

Table 4: Running times (in seconds) for the selected SDP benchmark problems for our solver PCSDP.

largest test problems CH4, hamming_9_5_5, hamming_10_2, hamming_11_2 and theta82. In Table 5 we therefore present only the running times in seconds when 4, 16, 32 and 64 processors were

Problem	m	n_{max}	*Phase*	4	8	16	32	64
CH4	24503	324	Elements	2407.36	1256.58	817.43	383.41	263.69
			Cholesky	11507.03	6581.86	3541.88	2963.46	1263.78
			Total	14987.14	8630.98	5060.78	3963.46	2113.39
hamming_9_5_6	53761	512	Elements	*	*	701.34	390.59	277.57
			Cholesky	*	*	108753.44	72082.25	26032.19
			Total	*	*	110908.28	73736.39	27570.8
hamming_10_2	23041	1024	Elements	535.18	324.41	242.94	160.87	136.19
			Cholesky	33259.07	18460.04	9007.85	5232.26	1952.26
			Total	35023.35	19908.98	10398.88	6460.06	3165.4
hamming_11_2	56321	2048	Elements	*	*	1632.33	1167.66	1056.20
			Cholesky	*	*	160248.71	94922.18	36750.65
			Total	*	*	171201.57	104380.48	46087.89
theta82	23872	400	Elements	473.78	261.15	161.91	80.49	55.58
			Cholesky	35116.75	17637.60	9436.25	5677.68	2045.98
			Total	36090.25	18283.34	9936.71	6067.4	2403.21

Table 5: Running times (in seconds) for the selected large-scale SDP benchmark problems for our solver PCSDP ('*' - means lack of memory)

used. At least 4 nodes were required to accommodate the Schur complement matrix for problems with around $24,000$ constraints like CH4, hamming_10_2 and theta82. They have also a small dimension of the primal and dual matrix variables, hence the parallel operations dominate. As a result, very good scalability is observed not only in computing M and its Cholesky factorization but on the total running time as well. For the truly large-scale problems hamming_9_5_6 and hamming_11_2, at least 16 nodes were needed due to the amount of memory required. They both have more than $50,000$ constraints and 32bit addressing would not be enough to address the elements of Schur complement matrix M. In this case $m \gg n$ and the solver efficiently solved them with a good scalability in terms of running times between 16 and 64 CPUs.

6 Efficiency comparison with other parallel SDP solvers

We compare the parallel efficiency of our SDP software PCSDP with two other distributed memory solvers, namely SDPARA by Yamashita *et al.* [27] and PDSDP by Benson [1]. To obtain the corresponding efficiency results for SDPARA and PDSDP we calculated the parallel efficiency using (5.15) and running times reported in [5] and [1], respectively. We selected only the large-scale benchmark problems with available solution time on one processor. The results are presented in Tables 7 and 8.

Table 7 presents parallel efficiencies for SDPARA using between 1 and 64 processors for control10, control11, theta5 and theta6. It shows that computation of the elements of the Schur complement matrix scales very well, while the Cholesky factorization and other computations scale poorly. Comparing with results from our code in Table 6 we see that SDPARA has much

Problem	m	n_{max}	*Phase*	1	2	4	8	16	32	64
control10	1326	100	Elements	1	0.97	0.90	0.80	0.55	0.41	0.36
			Cholesky	1	0.72	0.44	0.28	0.20	0.11	0.07
			Total	1	0.88	0.69	0.49	0.30	0.17	0.09
control11	1596	110	Elements	1	1.05	0.98	0.86	0.57	0.50	0.35
			Cholesky	1	0.93	0.60	0.40	0.29	0.16	0.12
			Total	1	0.97	0.79	0.58	0.35	0.23	0.13
theta5	3028	250	Elements	1	0.82	0.52	0.47	0.34	0.30	0.17
			Cholesky	1	0.97	0.84	0.61	0.52	0.30	0.26
			Total	1	0.89	0.72	0.50	0.37	0.21	0.14
theta6	4375	300	Elements	1	1.01	0.80	0.83	0.59	0.43	0.33
			Cholesky	1	0.95	0.92	0.72	0.65	0.45	0.37
			Total	1	0.95	0.86	0.71	0.52	0.29	0.22
theta8	7905	400	Elements	1	0.94	0.91	0.84	0.65	0.47	0.29
			Cholesky	1	0.70	0.65	0.80	0.76	0.43	0.52
			Total	1	0.71	0.64	0.72	0.62	0.35	0.31
theta42	5986	200	Elements	1	1.01	0.97	0.91	0.69	0.65	0.36
			Cholesky	1	0.90	0.82	0.82	0.76	0.43	0.48
			Total	1	0.90	0.81	0.80	0.67	0.39	0.35
theta62	13390	300	Elements	1	0.99	0.85	0.88	0.73	0.74	0.54
			Cholesky	1	0.58	0.56	0.69	0.66	0.51	0.66
			Total	1	0.59	0.56	0.68	0.63	0.48	0.54
thetaG51	6910	1001	Elements	1	1.00	0.98	0.98	0.91	0.87	0.72
			Cholesky	1	0.79	0.71	0.79	0.75	0.44	0.50
			Total	1	0.72	0.53	0.39	0.24	0.12	0.07
maxG51	1000	1000	Elements	1	0.59	0.52	0.58	0.42	0.34	0.26
			Cholesky	1	0.63	0.41	0.22	0.13	0.06	0.03
			Total	1	0.50	0.25	0.12	0.06	0.03	0.02
hamming_8_3_4	16129	256	Elements	1	1.02	0.84	0.82	0.68	0.72	0.51
			Cholesky	1	0.72	0.56	0.56	0.63	0.55	0.70
			Total	1	0.88	0.68	0.68	0.74	0.64	0.67
fap09	15225	174	Elements	1	0.94	0.67	0.68	0.48	0.50	0.31
			Cholesky	1	0.58	0.61	0.61	0.59	0.50	0.65
			Total	1	0.60	0.61	0.61	0.57	0.47	0.53

Table 6: Parallel efficiencies for the selected SDP problems for our solver PCSDP.

better scaling for control11, theta5 and theta6 test problems for *Elements*. SDPARA obtains a parallel efficiency $E > 1$ when computing *Elements* for the problems control10 and control11. These anomalities can occur because of cache issues when adding more processors. There can also be a difference in the number of iterations required by the algorithm when run on different

Problem	m	n_{max}	*Phase*	1	2	4	8	16	32	64
control10	1326	100	Elements	1.00	1.02	1.01	1.04	1.09	1.06	1.03
			Cholesky	1.00	0.87	0.65	0.39	0.31	0.16	0.13
			Total	1.00	0.94	0.85	0.74	0.65	0.43	0.28
control11	3028	250	Elements	1.00	1.03	1.03	1.02	1.05	1.05	1.05
			Cholesky	1.00	0.93	0.73	0.44	0.34	0.19	0.16
			Total	1.00	0.94	0.88	0.76	0.64	0.50	0.34
theta5	4375	300	Elements	1.00	1.11	1.17	1.21	1.25	1.21	1.14
			Cholesky	1.00	1.20	0.98	0.68	0.59	0.35	0.32
			Total	1.00	0.97	0.77	0.54	0.40	0.27	0.17
theta6	1596	110	Elements	1.00	1.12	1.19	1.23	1.26	1.24	1.17
			Cholesky	1.00	1.29	1.12	0.83	0.73	0.48	0.38
			Total	1.00	0.88	0.89	0.67	0.55	0.37	0.25

Table 7: Parallel efficiencies for the selected SDP problems solved by SDPARA. Times were taken from [27].

Problem	m	n_{max}	*Phase*	1	2	4	8	16	32
control10	1326	100	Elements	1	0.98	0.81	0.69	0.67	0.61
			Cholesky	1	0.75	0.58	0.34	0.18	0.09
			Total	1	0.94	0.75	0.60	0.49	0.35
control11	1596	110	Elements	1	0.99	0.82	0.70	0.73	0.64
			Cholesky	1	0.76	0.64	0.37	0.22	0.11
			Total	1	0.95	0.78	0.63	0.58	0.42
theta42	5986	200	Elements	1	0.99	0.48	0.43	0.36	0.30
			Cholesky	1	0.86	0.77	0.63	0.51	0.36
			Total	1	0.88	0.70	0.58	0.47	0.34
theta6	4375	300	Elements	1	1.02	0.58	0.51	0.41	0.33
			Cholesky	1	0.83	0.72	0.56	0.44	0.29
			Total	1	0.87	0.67	0.53	0.40	0.27
theta8	7905	400	Elements	1	0.96	0.50	0.49	0.41	0.36
			Cholesky	1	0.88	0.78	0.69	0.58	0.41
			Total	1	0.89	0.69	0.62	0.51	0.37
maxG51	1000	1000	Elements	1	0.99	0.78	0.73	0.63	0.49
			Cholesky	1	0.73	0.50	0.28	0.15	0.07
			Total	1	0.81	0.56	0.36	0.21	0.10

Table 8: Parallel efficiencies for the selected SDP problems solved by PDSDP. Times were taken from [1].

numbers of processors due to numerical rounding issues. Another factor that influences the scalability is the choice of the distributed layout for computing *Elements*. It is known that the best load balance is achieved using one-durational cyclic row distribution as in SDPARA. Unfortunately this has to be redistributed later on to a two-dimensional block cyclic distribution which means a great deal of network communication. Especially, if the size of Schur complement matrix is very large, *i.e.* $m \gg n$, and the network is not a low latency one, the result is higher running times and worse total time scalability.

Analyzing the times of Cholesky factorization of the Schur complement matrix we see that the different scalability values are much closer there. However, the advantage is still on the side of SDPARA. In fact, we would expect our solver PCSDP to show better scalability on a truly large scale problems. All four problems in Table 7 are medium sized.

Similarly, we computed parallel efficiencies for the problems solved with 1 to 32 processors by PDSDP. (Data for the running times on a 64 CPU's was not available in [1].) The results are shown in Table 8, as calculated from (5.15).

There was no data for problems theta5 or theta62, so they are not present in the table. The overall parallel efficiency is very close for both PCSDP and PDSDP. PDSDP has better scalability when computing *Elements* for the smaller problems control10 and control11. The reason is the use of a one-dimensional block cyclic distribution. It obviously works well in terms of load balance on small and medium sized problems. This is not the case on larger size problems as theta6 and theta42. In those cases PCSDP shows consistently better scalability. We already mentioned the issues with maxG51 and the dual-scaling algorithm manages to achieve better scaling for this problem. If we compare computing the Cholesky factorization, then our code has better scalability than PDSDP except for the problems control10 and theta5. On larger problems PCSDP consistently shows better scalability results. The total running time for control10 and control11 are worse for our implementation than for PDSDP. Low scalability of *Elements* on these moderate sized SDP problems causes worse total time efficiency. On the large scale problems theta6 and theta42 PCSDP completely outperforms PDSDP regardless of the number of processors.

7 Concluding Remarks

In this paper we presented a parallel implementation of a 64 bit version of the primal-dual IPM solver CSDP for SDP problems for Beowulf clusters. The software makes use of a two-dimensional block cyclic data distribution to compute the Schur complement matrix. A new feature is implemented to exploit rank-one structure in the constraint matrices. It still is unable to perform as fast as the dual-scaling IPM code DSDP on a rank-one problems, but it decreases the time gap in practice by a factor of two.

Our parallel implementation comes into its own for problems with very large numbers of linear constraints ($m > 20,000$) and moderately sized matrix variables $n \ll m$. For such problems very good parallel speedup is observed.

Name	m	n	n_{max}	Optimal value
control10	1326	150	100	$3.8533E+01$
control11	1596	165	110	$3.1959E+01$
theta5	3028	250	250	$5.723231E+01$
theta6	4375	300	300	$6.347709E+01$
theta8	7905	400	400	$7.3953559E+01$
theta42	5986	200	200	$2.3931707E+01$
theta62	13390	300	300	$2.9641248E+01$
theta82	23872	400	400	$3.4366889E+1$
thetaG51	6910	1001	1001	$3.49000E+02$
maxG51	1000	1000	1000	$4.003809E+03$
hamming_8_3_4	16129	256	256	$2.560000e+01$
hamming_9_5_6	53761	512	512	$8.5333332E+01$
hamming_10_2	23041	1024	1024	$1.024E+02$
hamming_11_2	56321	2048	2048	$1.7066666E+02$
fap09	15225	174	174	$-1.0797803E+01$
CH4.1A1.STO6G	24503	630	324	$1.3021808E+01$

Table 9: Selected SDP test problems.

Appendix: SDP Benchmark Problems

References

[1] S. J. Benson. Parallel computing on semidefinite programs. Technical Report ANL/MCS-P939-0302, Mathematics and Computer Science Division, Argonne National Laboratory, April 2003.

[2] S. J. Benson, Y. Ye, and X. Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. *SIAM J. Optim.*, 10(2):443–461 (electronic), 2000.

[3] B. Borchers. CSDP, a C library for semidefinite programming. *Optim. Methods Softw.*, 11/12(1-4):613–623, 1999.

[4] B. Borchers. SDPLIB 1.2, library of semidefinite programming test problems. *Optim. Methods Softw.*, 11/12(1-4):683–690, 1999.

[5] B. Borchers and J. Young. Implementation of a primal-dual method for SDP on a shared memory parallel architecture. To Appear in *Computational Optimization and Applications.*

[6] E. de Klerk. *Aspects of semidefinite programming, Interior point algorithms and selected applications*, volume 65 of *Applied Optimization*. Kluwer Academic Publishers, Dordrecht, 2002.

[7] E. de Klerk, G. Elabwabi, and D. den Hertog. Optimization of univariate functions on bounded intervals by interpolation and semidefinite programming. CentER Discussion paper 2006-26, Tilburg University, The Netherlands, April 2006. Available at *Optimization Online*.

[8] E. de Klerk, C. Roos, and T. Terlaky. Initialization in semidefinite programming via a self-dual skew-symmetric embedding. *Oper. Res. Lett.*, 20(5):213–221, 1997.

[9] G. Elabwabi. *Topics in global optimization using semidefinite optimization.* PhD thesis, Delft University of Technology, to be published (2007).

[10] K. Fujisawa, M. Kojima, and K. Nakata. Exploiting sparsity in primal–dual interior-point methods for semidefinite programming. *Math. Programming*, 79(1-3, Ser. B):235–253, 1997.

[11] M. Fukuda, M. Kojima, K. Mucro, and K. Nakata. Exploiting sparsity in semidefinite programming via matrix completion. I. General framework. *SIAM J. Optim.*, 11(3):647–674 (electronic), 2000/01.

[12] P. Hansen, B. Jaumard, and S.-H. Lu. Global optimization of univariate Lipschitz functions. II. New algorithms and computational comparison. *Math. Programming*, 55(3, Ser. A):273–292, 1992.

[13] C. Helmberg and F. Rendl. Solving quadratic $(0, 1)$-problems by semidefinite programs and cutting planes. *Math. Programming*, 82(3, Ser. A):291–315, 1998.

[14] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM J. Optim.*, 6(2):342–361, 1996.

[15] M. Kojima, S. Shindoh, and S. Hara. Interior-point methods for the monotone semidefinite linear complementarity problem in symmetric matrices. *SIAM J. Optim.*, 7(1):86–125, 1997.

[16] J. Löfberg and P. Parrilo. From coefficients to samples: a new approach to SOS optimization. In *IEEE Conference on Decision and Control*, December 2004.

[17] H. D. Mittelman. Dimacs challenge benchmarks. `http://plato.asu.edu/dimacs/`, 2000.

[18] R. D. C. Monteiro. Primal-dual path-following algorithms for semidefinite programming. *SIAM J. Optim.*, 7(3):663–678, 1997.

[19] R. D. C. Monteiro and P. Zanjácomo. Implementation of primal-dual methods for semidefinite programming based on Monteiro and Tsuchiya Newton directions and their variants. *Optim. Methods Softw.*, 11/12(1-4):91–140, 1999.

[20] K. Nakata, K. Fujisawa, M. Fukuda, M. Kojima, and K. Mucro. Exploiting sparsity in semidefinite programming via matrix completion. II. Implementation and numerical results. *Math. Program.*, 95(2, Ser. B):303–327, 2003.

[21] M. Nakata, H. Nakatsuji, M. Ehara, M. Fukuda, K. Nakata, and K. Fujisawa. Variational calculations of fermion second-order reduced density matrices by semidefinite programming algorithm. *The Journal of Chemical Physics*, 114(19):8282–8292, 2001.

[22] Y. Nesterov. Squared functional systems and optimization problems. In *High performance optimization*, volume 33 of *Appl. Optim.*, pages 405–440. Kluwer Acad. Publ., Dordrecht, 2000.

[23] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optim. Methods Softw.*, 11/12(1-4):625–653, 1999.

[24] K.-C. Toh and M. Kojima. Solving some large scale semidefinite programs via the conjugate residual method. *SIAM J. Optim.*, 12(3):669–691 (electronic), 2002.

[25] K. C. Toh, M. J. Todd, and R. H. Tütüncü. SDPT3—a MATLAB software package for semidefinite programming, version 1.3. *Optim. Methods Softw.*, 11/12(1-4):545–581, 1999.

[26] M. Yamashita, K. Fujisawa, and M. Kojima. Implementation and evaluation of SDPA 6.0 (semidefinite programming algorithm 6.0). *Optim. Methods Softw.*, 18(4):491–505, 2003.

[27] M. Yamashita, K. Fujisawa, and M. Kojima. SDPARA: SemiDefinite Programming Algorithm paRAllel version. *Parallel Comput.*, 29(8):1053–1067, 2003.